
Message Passing Programming Model

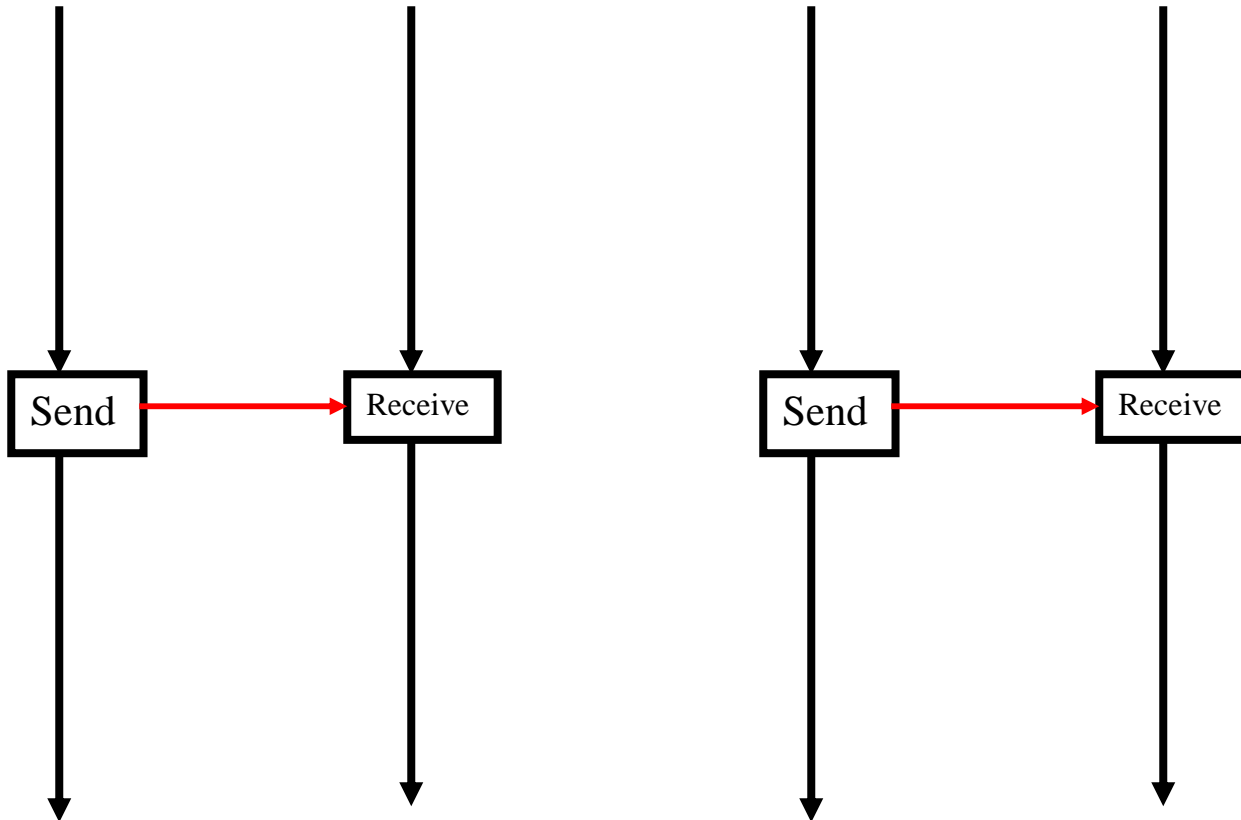
AMANO, Hideharu

Textbook pp.140—147

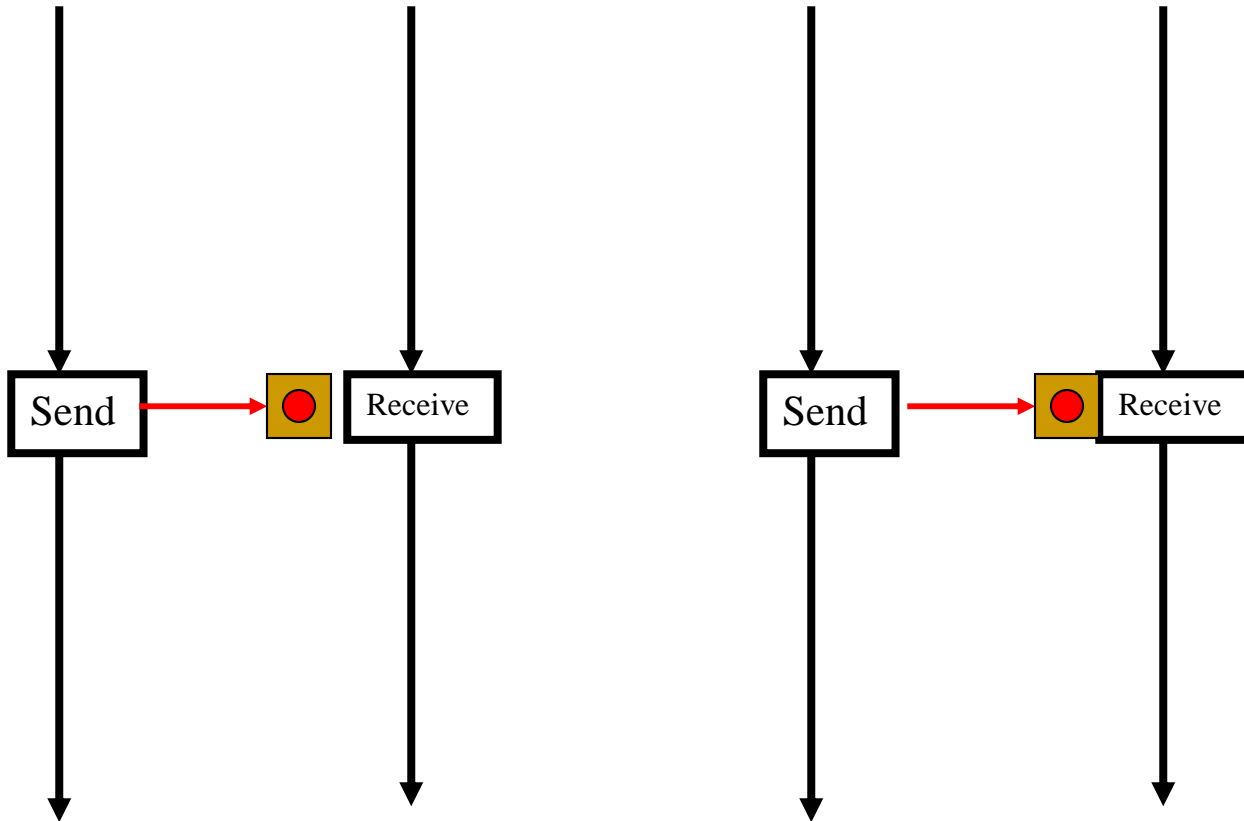
Message Passing Model

- No shared memory
 - Easy to be implemented in any parallel machines
 - Popularly used for PC Clusters
 - Today, we focus on MPI.
-

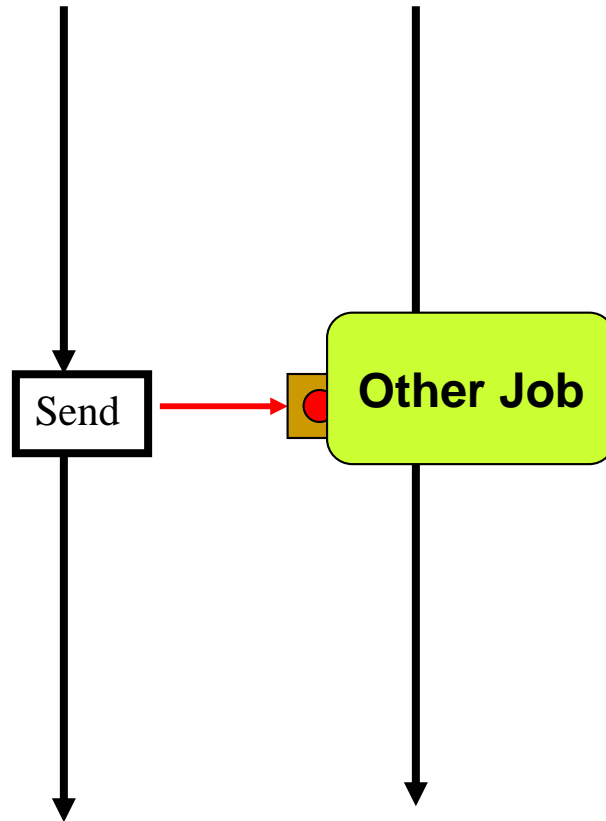
Message passing (Blocking: rendezvous)



Message passing (with buffer)



Message passing (non-blocking)



PVM (Parallel Virtual Machine)

- A buffer is provided for a sender.
 - Both blocking/non-blocking receive is provided.
 - Barrier synchronization
-

MPI

(Message Passing Interface)

- Superset of the PVM for 1 to 1 communication.
 - Group communication
 - Various communication is supported.
 - Error check with communication tag.
 - Detail will be introduced later.
-

Programming style using MPI

- SPMD (Single Program Multiple Data Streams)
 - Multiple processes executes the same program.
 - Independent processing is done based on the process number.
 - Program execution using MPI
 - Specified number of processes are generated.
 - They are distributed to each node of the NORA machine or PC cluster.
-

Communication methods

- Point-to-Point communication
 - A sender and a receiver executes function for sending and receiving.
 - Each function must be strictly matched.
 - Collective communication
 - Communication between multiple processes.
 - The same function is executed by multiple processes.
 - Can be replaced with a sequence of Point-to-Point communication, but sometimes effective.
-

Fundamental MPI functions

- Most programs can be described using six fundamental functions
 - `MPI_Init()` ... MPI Initialization
 - `MPI_Comm_rank()` ... Get the process #
 - `MPI_Comm_size()` ... Get the total process #
 - `MPI_Send()` ... Message send
 - `MPI_Recv()` ... Message receive
 - `MPI_Finalize()` ... MPI termination
-

Other MPI functions

- Functions for measurement
 - `MPI_Barrier()` ... barrier synchronization
 - `MPI_Wtime()` ... get the clock time
 - Non-blocking function
 - Consisting of communication request and check
 - Other calculation can be executed during waiting.
-

A simple example: Hollow

```
1: #include <stdio.h>
2: #include <mpi.h>
3:
4: #define MSIZE 64
5:
6: int main(int argc, char **argv)
7: {
8:     char msg[MSIZE];
9:     int pid, nprocs, i;
10:    MPI_Status status;
11:
12:    MPI_Init(&argc, &argv);
13:    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
14:    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
15:
16:    if (pid == 0) {
17:        for (i = 1; i < nprocs; i++) {
18:            MPI_Recv(msg, MSIZE, MPI_CHAR, i, 0, MPI_COMM_WORLD, &status);
19:            fputs(msg, stdout);
20:        }
21:    }
22:    else {
23:        sprintf(msg, "Hello, world! (from process #%d)\n", pid);
24:        MPI_Send(msg, MSIZE, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
25:    }
26:
27:    MPI_Finalize();
28:
29:    return 0;
30: }
```

Initialization and termination

```
int MPI_Init(  
    int *argc, /* pointer to argc */  
    char ***argv /* pointer to argv */ );
```

argc and argv come from command line like common C programming

```
int MPI_Finalize();
```

Example:

```
MPI_Init (&argc, &argv);
```

```
...
```

```
MPI_Finalize();
```

Communicators: a space for communication

MPI_COMM_WORLD is a communicator for all processes.

```
int MPI_Comm_rank(
    MPI_Comm comm, /* communicator */
    int *rank /* process ID (output) */ );           //Return process ID (rank)

int MPI_Comm_size(
    MPI_Comm comm, /* communicator */
    int *size /* number of process (output) */ ); //Return the number of all processes.
```

Example:

```
int pid, nproc;
MPI_Comm_rank(MPI_COMM_WORLD, &pid); // My process id
MPI_Comm_rank(MPI_COMM_WORLD, &nproc); // Total processor number
```

MPI_Send

1 to 1 message send

```
int MPI_Send(  
    void *buf, /* send buffer */  
    int count, /* # of elements to send */  
    MPI_Datatype datatype, /* datatype of elements */  
    int dest, /* destination (receiver) process ID */  
    int tag, /* tag */  
    MPI_Comm comm /* communicator */ );
```

```
MPI_Send(msg, MSIZE, MPI_CHAR, 0,0, MPI_COMM_WORLD);
```

Send MSIZE characters in the array “msg” to process 0 with tag 0.

MPI_Recv which matches the tag can receive the message.

MPI_Recv

1 to 1 message receive

```
int MPI_Recv(  
    void      *buf,           /* receiver buffer */  
    int       count,         /* # of elements to receive */  
    MPI_Datatype datatype,   /* datatype of elements */  
    int       source,        /* source (sender) process ID */  
    int       tag,           /* tag */  
    MPI_Comm comm,          /* communicator */  
    MPI_Status /* status (output) */ );
```

```
char msg[MSIZE]  
MPI_Status status;
```

```
MPI_Recv(msg, MSIZE, MPI_CHAR, 1, 0, MPI_COMM_WORLD, &status);  
fputs(msg, stdout);
```

Receive MSIZE characters from process 1 with tag 0, and store in the array “msg”.

- “status” shows the status of receiving message.
-

MPI_Bcast

Broadcast to all processes

```
int MPI_Bcast(  
    void *buf, /* send buffer */  
    int count, /* # of elements to send */  
    MPI_Datatype datatype, /* datatype of elements */  
    int root, /* Root processor number */  
    MPI_Comm comm /* communicator */ );  
  
if (pid ==0)  
    a=1.0;  
  
MPI_Bcast(&a,1,MPI_DOUBLE, 0, MPI_COMM_WORLD);  
  
pid 0 sends "a" to all processes.
```

datatype and count

- The size of the message is identified with count and datatype.
 - MPI_CHAR char
 - MPI_INT int
 - MPI_FLOAT float
 - MPI_DOUBLE double ... etc.
-

Compile and Execution

```
% mpicc -o hello hello.c
```

```
% mpirun -np 4 ./hello
```

```
Hello, world! (from process #1)
```

```
Hello, world! (from process #2)
```

```
Hello, world! (from process #3)
```

reduct.c: Initialize

```
int pid, nproc, i;
FILE *fin;
double mat[N];
double sum, psum;
double start, startcomp, end;
MPI_Status status;
    if((fin = fopen("mat4k.dat", "r"))==NULL) {
        fprintf(stderr, "mat.dat is not existing¥n");
        exit(1);
    }
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &pid);
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
```

mat4k.dat has the
data

MPI Initialize

reduct.c: host (pid=0)

```
sum=0.0;
```

```
if (pid == 0) {
```

```
for (i = 0; i < N; i++) { fscanf(fin, "%lf", &mat[i]); }
```

```
start = MPI_Wtime();
```

```
for (i = 1; i < nproc; i++)
```

```
MPI_Send(&mat[i*N/nproc], N/nproc, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
```

```
startcomp = MPI_Wtime();
```

```
for(i = 0; i < N/nproc; i++) sum += mat[i];
```

```
for (i = 1; i < nproc; i++) {
```

```
MPI_Recv(&psum, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &status);
```

```
sum += psum;
```

```
}
```

```
end = MPI_Wtime();
```

```
printf("%lf¥n", sum);
```

```
printf("Total time = %lf Exect time= %lf [sec]¥n", end-start, end-startcomp);
```

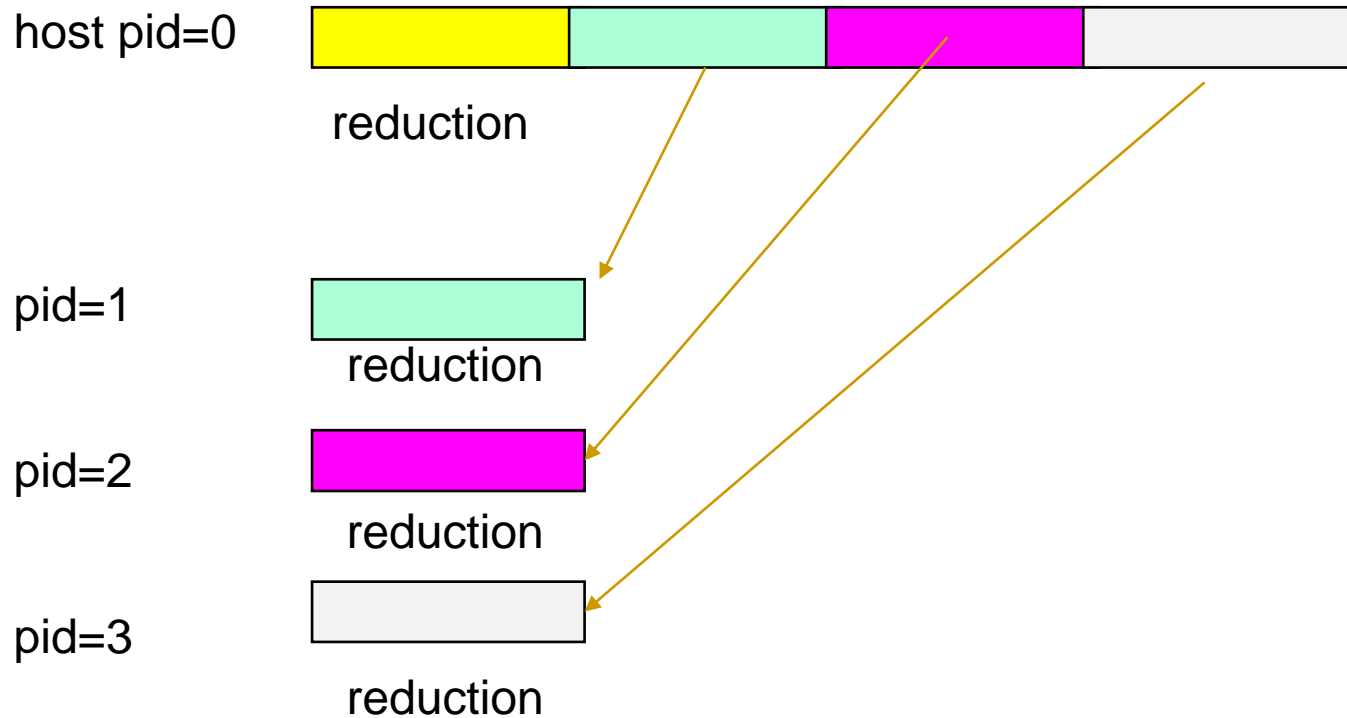
```
}
```

Read data from
the file

MPI send

Compute its own
part.

Distribution of data in the array



reduct.c: slave processors

```
else {
    i=0;
    MPI_Recv(&mat[i], N/nproc, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
&status);
    for(i = 0; i < N/nproc; i++) sum += mat[i];
    MPI_Send(&sum, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
}

MPI_Finalize();

return 0;
}
```

Receive
data

Partial sum

Send the
result
to host

Excise

- Assume that there is an array of coefficient $x[4096]$.
- Write the MPI code for computing sum of square of difference of all combinations.

```
sum = 0.0;
```

```
for (i=0; i<N; i++)
```

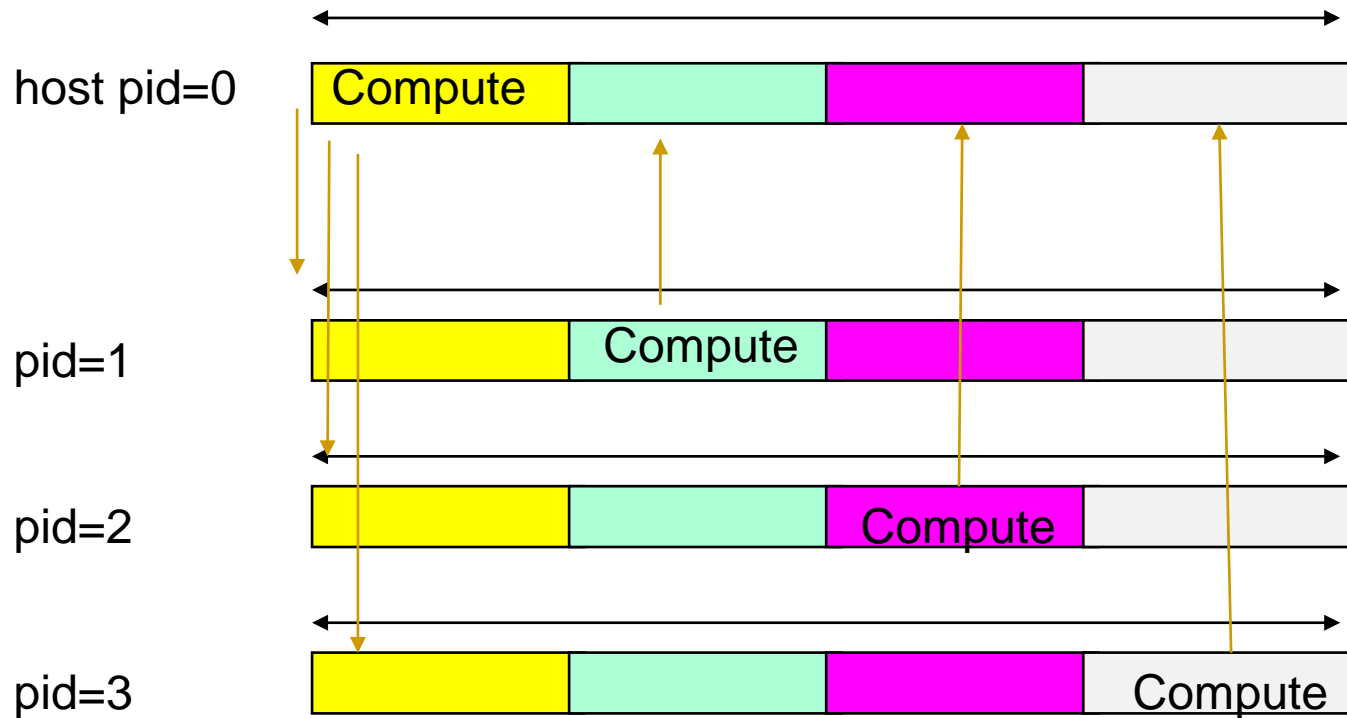
```
    for(j=0; j<N; j++)
```

```
        sum += (x[i]-x[j])*(x[i]-x[j]);
```


Hint

- **Distribute x to all processors.**
- **Each processor computes partial sums.**
sum=0.0;
for (i=N/nproc*pid; i<N/nproc*(pid+1); i++)
 for(j=0; j<N; j++)
 sum += (x[i]-x[j])*(x[i]-x[j]);
- **Then, send sum to processor 0.**
- **Execution time with 1,2,4 processors is evaluated.**
- **Note that the computation results are not exactly the same.**
- **By using MPI_Bcast, the performance is slightly improved.**

Distribution of data in the array



Distribute the whole array
Compute only a part
Return the computed part

Report

- Submit the followings:
 - MPI C source code

